Moving towards automated game play-testing

Celso Gabriel Malosto[1], Luciana Campos[1], Igor de Oliveira Knop[1]

[1]Grupo de Educação Tutorial em Sistemas de Informação
Universidade Federal de Juiz de Fora Juiz de Fora – MG – Brasil

gabriel.malosto@estudante.ufjf.br, {igor.knop, luciana.campos}@ufjf.br

***Abstract. Introduction**: Prototyping games requires multiple testing phases, some of which focus on balancing rules and keeping the game fun. These are typically done by human testers, who can influence the results while requiring a significant time investment. **Objective:** This work in progress reports the implementation of a framework for stress testing games in the development phase. **Steps:** The data for decision-making is generated by intelligent agents trained in an AlphaZero-inspired method, which aligns residual neural networks with the Monte-Carlo Tree Search algorithm. **Expected results:** Game designers should be able to describe their game on our platform. The process of training and using agents will allow us to capture eventual balancing problems and dominant strategies, providing valuable information to guide changes in the rules, thus improving the player experience.*
***Keywords** game development, play-testing, AlphaZero, residual neural networks, Monte-Carlo tree search.*

1. Introduction

Games are structured activities with clearly defined objectives. A player wins by reaching the goal while adhering to a set of predetermined conditions. These rules allow for diverse strategies, which can be evaluated considering the context of the match [Suits 1967]. Among their categories, turn-based games stand out, in which time progresses in discrete steps. During each turn, a player can take a limited set of actions, each altering the game's state. Players take alternating turns until the game reaches its terminal state. As a result, the game's progression can be represented as a graph, where nodes correspond to states and edges represent possible actions [Salen e Zimmerman 2003].

These data structures prove valuable during game prototyping, enabling designers to analyze gameplay dynamics and detect potential design flaws. Balance adjustments require careful contextual consideration, addressing factors such as: mathematical equilibrium, content and difficulty progression, strategic diversity, and inter-player fairness [Romero e Schreiber 2021]. Play-testing is a design process where testers simulate gameplay to explore the behavior of systems [Fullerton 2019]. Traditionally conducted by human testers, this approach presents limitations: it is time-consuming and can be influenced by their expectations, moods, or lack of concentration. While these elements naturally affect player experiences in finished products, they introduce unwanted variables, particularly for stress testing and game balancing [Marcelo e Pescuite 2009].

This study follows our exploratory research [Araki e Knop 2020, Malosto et al. 2023] into automating play-testing processes through intelligent agents. We propose a framework for generating analytical insights from simulation histories

to support design decisions. It works by modeling turn-based game prototypes, which allows for the training of agents to autonomously explore the game space. The framework's development requires formal representations of game concepts (states, moves, slots, players), as well as the integration of search algorithms and reinforcement learning methods. This paper is organized as follows: Section 2 establishes the theoretical foundation; Section 3 reviews relevant literature; Section 4 outlines the methodological approach and implementation steps; Section 5 describes expected results; and Section 6 concludes with final remarks and study limitations.

2. Theoretical basis

We investigated the Monte Carlo Tree Search (MCTS) method, which is a decision-making algorithm that represents gameplay as a tree structure [Kocsis e Szepesvári 2006, Coulom 2006]. The root node contains the initial game state. Edges represent transitions between a state and its possible successors. Each subsequent level alternates between perspectives of each player. This structure allows the algorithm to explore opponent moves and suggest optimal future actions [Świechowski et al. 2022].

The search starts by (1) traversing the tree guided by the Upper Confidence Bounds (UCB) policy [Kocsis e Szepesvári 2006] to select a not-expanded leaf node. This balances exploration and exploitation, keeping track of each node's victory and visit counts. Next, (2) expansion applies a valid action to the node, generating its successor state. Then, (3) simulation continues the match until the game is over. Finally, (4) backpropagation updates the tracked counts for all nodes along the selection path.

MCTS has been combined with Residual Neural Networks (ResNets) to eliminate the simulation step [He et al. 2015]. These Artificial Intelligence (AI) models originate from Convolutional Neural Networks (CNNs) and were initially developed for image recognition tasks [Li et al. 2022]. Their architecture comprises blocks of successive convolutional layers and normalization operations, employing Rectified Linear Unit (ReLU) activation functions as described in [Nair e Hinton 2010].

Building on this architecture, AlphaZero was developed for general board game play. The system takes the game state as input and produces two outputs: (1) a probability tensor over possible actions, and (2) a scalar value estimating the score of the expected outcome. AlphaZero employs a reinforcement learning algorithm for network training, where the agent improves by playing self-matches through a process called self-play [Silver et al. 2016, Silver et al. 2017, Silver et al. 2018].

3. Related work

[Zook et al. 2019] highlight the advantages of replacing human players in specific parts of the play-testing process. They combine regression and classification techniques to perform active learning [Cohn et al. 1994] in a *shoot'em up* game. The game's mechanics are well-defined, but parameters such as player, enemy, and bullet speeds are adjusted through exhaustive testing, which is here replaced by automated play-testing.

In the works of [Gudmundsson et al. 2018] and [Zook et al. 2019], MCTS is used alongside CNNs. These are trained on a massive dataset of real players to predict quest difficulty in digital *match-3* games—respectively, *Candy Crush* and *Jewels Star Story*.

While existing play-testing research predominantly examines digital games (typically modeled as continuous-time systems), we argue these techniques can be effectively adapted for physical games (modeled as discrete systems).

4. Steps

This work introduces Auto Play-Test System (APTS), a framework for automating the play-testing process in turn-based games. The system requires designers to formally specify game rules, while allowing for modification of key parameters including: scoring mechanisms, valid move generation, reaching of terminal conditions, etc. The engine subsequently generates intelligent agents that engage in self-play to produce analytical data for design evaluation. The framework's primary objective is to provide quantitative insights to inform game balancing decisions.

Game prototypes must be implemented as independent modules from the core application engine. As illustrated in Figure 1, all implementations must inherit from and conform to our base abstract classes. The fundamental `Game` class defines the game rules through its abstract methods, requiring developers to specify: (1) the complete set of valid moves and (2) the players' configuration. These elements must be declared during initialization as they serve as critical references for the classification neural network.

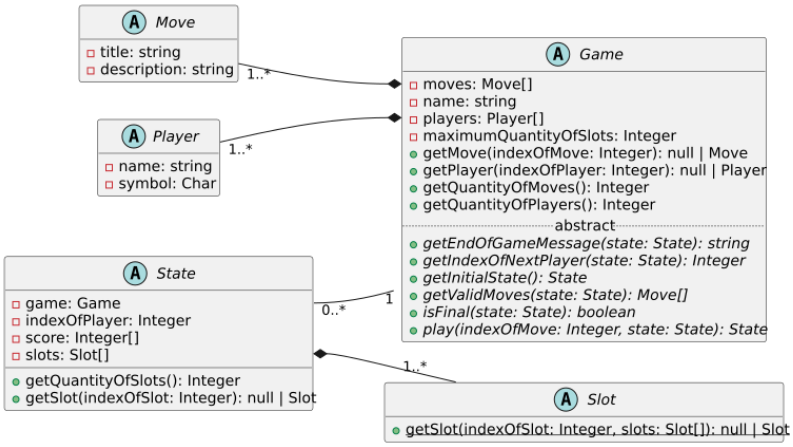

Figure 1. Class diagram representing required classes for describing a game: Game, State, Slot, Move and Player. Created by the authors (2025).

The game process begins by generating an initial state. During each turn, the system first identifies all valid moves available in the current state, then selects one according to the game's decision-making logic. The chosen move is applied to create a new game state, triggering updates to the game score and player slots. After each move application, the system evaluates whether the new state meets terminal conditions. When a terminal state is reached, the game concludes by displaying the final results. This sequence continues iteratively until the termination criteria are satisfied.

The decision-making process is managed by the MCTS algorithm operating on a given game state. The algorithm generates a probability distribution over all possible moves, represented as an array. Each element contains the computed selection probability for its corresponding move, with higher values indicating more favorable actions according to the algorithm's evaluation.

An intelligent agent is fundamentally characterized by its ResNet architecture, which can be dynamically configured per game prototype with variable network depth and input shape, implemented using the library TensorFlow.js [Abadi et al. 2015]. The model's input tensor must encode all relevant information about a state. During gameplay, the agent's MCTS algorithm transforms the current state into the required tensor representation and processes it through the neural network. It receives two outputs: (1) a probability distribution tensor over all moves of the game and (2) a value estimate predicting the expected match outcome from the current state.

Taking Tic-Tac-Toe[1] as a concrete example, the state in Figure 2 is encoded similarly to an RGB image: the first player's (X) occupied positions are encoded in the blue channel (marked as 1 if present, 0 otherwise), the second player's (O) positions are encoded in the red channel, while empty board slots populate the green channel.

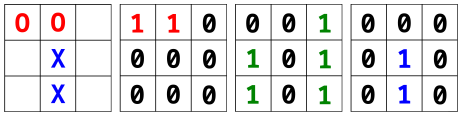

Figure 2. Each content of the board to the left is mapped to a channel, as shown in the following matrices. Created by [Malosto et al. 2023].

The MCTS algorithm uses the valid moves array to filter the move probabilities, allowing it to expand the search tree to all legal moves without performing simulations. It stores the neural network's predicted value (scalar) at each expanded node. To guide this process, we employ a modified UCB policy that incorporates the scalar to bias the exploration toward more promising branches.

To train an intelligent agent, we create a training dataset through self-play sessions where the agent competes against itself. During each game turn, we record: (1) the encoded game state, (2) the move probability distribution generated by MCTS, and (3) the final match result (win, loss, or draw). The collected data is used to adjust the model's parameters using the Adaptive Moment Estimation (Adam) optimizer.

5. Expected results

Different versions of the engine have been developed, targeting some of the proposed objectives, all of which can be accessed via the GitHub public repository[2]. We have been able to implement the development cycle for Tic-Tac-Toe and Connect Four[3], which covers: (1) representing the game, (2) generating an intelligent agent, and (3) training it to play against itself or a human player. This was exposed both via a command-line interface and a web-based graphical interface. However, due to the simplistic nature of these games, the current implementation cannot be used to represent any arbitrary game prototype. More complex and commercially available games, such as Gobblet Gobblers[4], Boop[5] and Checkers[6] have been implemented, although only using the MCTS method.

[1]Available at `https://boardgamegeek.com/boardgame/11901/tic-tac-toe`.
[2]Available at `https://github.com/ufjf-gamelab/apts`.
[3]Available at `https://boardgamegeek.com/boardgame/2719/connect-four`.
[4]Available at `https://boardgamegeek.com/boardgame/13230/gobblet-gobblers`.
[5]Available at `https://boardgamegeek.com/boardgame/355433/boop`.
[6]Available at `https://boardgamegeek.com/boardgame/149805`.

To achieve this generalization, we re-implemented the engine developed at [Malosto et al. 2023] based on the abstractions defined in Figure 1. The ongoing development includes unit tests to ensure the correct implementation of the game rules. We expect to refactor the MCTS and ResNet modules soon, which will allow for deploying and training generic agents. The monorepo structure facilitates testing via the command line while ensuring the system can be deployed as a web application.

Future goals include using a domain specific description language, such as Game Description Language (GDL)[7] or Zillions by Rules Files (ZRF)[8] to define the game rules, which currently must be implemented on source-code. We also aim to investigate methods for forking existing prototypes to generate modified versions without necessitating full retraining from scratch. Doing so requires a more in-depth understanding of the representation of the game state as a tensor, which can make the neural network incompatible between versions if structural changes are done.

Most fundamentally, it is necessary to define relevant metrics to collect about the games in testing and implement this feature into the self-play process, that currently only collects the game history. The goal is to provide designers with a comprehensive analysis of the game, allowing them to compare different versions and identify the most promising ones for further development. In order to evaluate the performance of the intelligent agent, we plan on conducting a series of experiments comparing its performance against human players. It is yet necessary to determine all relevant metrics to be collected and analyzed.

6. Final considerations

This work presents a framework for automating the play-testing process in turn-based games. The system aims to generate intelligent agents that autonomously explore the game space of prototypes and produce analytical data to support design decisions. We hope to contribute to the field of game design and development by reducing the need for human testers in the early stages of game design, in which the game is still being defined and the rules are not yet fully established. The framework's development is ongoing, with the current focus on implementing a generic engine capable of representing arbitrary turn-based games. The initial implementation has been successfully tested with Tic-Tac-Toe and Connect Four, demonstrating the feasibility of the approach.

The next steps involve refining the engine, enhancing the MCTS and ResNet modules, and integrating a domain-specific description language. The ultimate goal is to provide a tool for automating stress tests in the play-testing process, enabling designers to make informed decisions based on data-driven insights. As limitations, we exclude continuous-time games from our scope, as they require different modeling approaches. We also recognize the need for further research on representing non-board games, such as card games, whose states cannot be as easily mapped to the input tensor.

7. Acknowledgments

This work was supported by the Grupo de Educação Tutorial em Sistemas de Informação (GET-SI) and Pró-Reitoria de Extensão (PROEX) at UFJF through research stipends for undergraduate students.

[7]GDL available at `http://logic.stanford.edu/ggp/notes/gdl.html`.
[8]ZRF available at `https://www.zillionsofgames.com/language`.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., e Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Araki, D. S. e Knop, I. O. (2020). Testes de software e simulações como ferramentas para game design. In *Brazilian Symposium on Computer Games and Digital Entertainment 2020 Proceedings*.

Cohn, D., Atlas, L., e Ladner, R. (1994). Improving generalization with active learning. *Machine learning*, 15:201–221.

Coulom, R. (2006). Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer.

Fullerton, T. (2019). *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*. CRC Press, Boca Raton, 4 edition.

Gudmundsson, S., Eisen, P., Poromaa, E., Nodet, A., Purmonen, S., Kozakowski, B., Meurling, R., e Cao, L. (2018). Human-like playtesting with deep learning. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE.

He, K., Zhang, X., Ren, S., e Sun, J. (2015). Deep residual learning for image recognition.

Kocsis, L. e Szepesvári, C. (2006). Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer.

Li, Z., Liu, F., Yang, W., Peng, S., e Zhou, J. (2022). A survey of convolutional neural networks: Analysis, applications, and prospects. *IEEE Transactions on Neural Networks and Learning Systems*, 33(12):6999–7019.

Malosto, C. G. D. A., Knop, I. O., e Conceição, L. D. C. (2023). Alphazero como ferramenta de playtest. *Revista ComInG - Communications and Innovations Gazette*, 7(1):39–50.

Marcelo, A. e Pescuite, J. (2009). *Design de jogos: Fundamentos*. Brasport, Rio de janeiro, 1 edition.

Nair, V. e Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 807–814, Madison, WI, USA. Omnipress.

Romero, B. e Schreiber, I. (2021). *Game Balance*. CRC Press, Boca Raton, 1st edition edition.

Salen, K. e Zimmerman, E. (2003). *Rules of Play: Game Design Fundamentals*. MIT Press, Cambridge.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., e Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., e Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., e Hassabis, D. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144.

Suits, B. (1967). What is a game? *Philosophy of Science*, 34(2):148–156.

Świechowski, M., Godlewski, K., Sawicki, B., e Mańdziuk, J. (2022). Monte carlo tree search: a review of recent modifications and applications. *Artificial Intelligence Review*, 56(3):2497–2562.

Zook, A., Fruchter, E., e Riedl, M. (2019). Automatic playtesting for game parameter tuning via active learning.